Universal Content Launcher

A series of tutorials detailing the creation of a content launcher in Microsoft Visual C#

Tutorial Four

Building in Features

Contents

Introduction 3
Tutorial Requirements 3
Part One: Expanding the Content Loader 4
Part Two: Creating an Add Content Form 14
Part Three: Adding records to the Content XML file 21

Introduction

In tutorial three, we finally managed to get to launch some content, having been through ArrayLists, Inheritance and Reflection all in one go! Now it's time to build on what we have, and start to add some of the features that are missing, like displaying content of all types, and adding the ability to manage the XML content file from within UCL.

Tutorial Requirements

These tutorials are sequential, so if you haven't already been through the previous tutorials, then please download, and work through them before continuing with this one. Tutorial one also outlines the software you will need for creating UCL, and where you can download it for free.

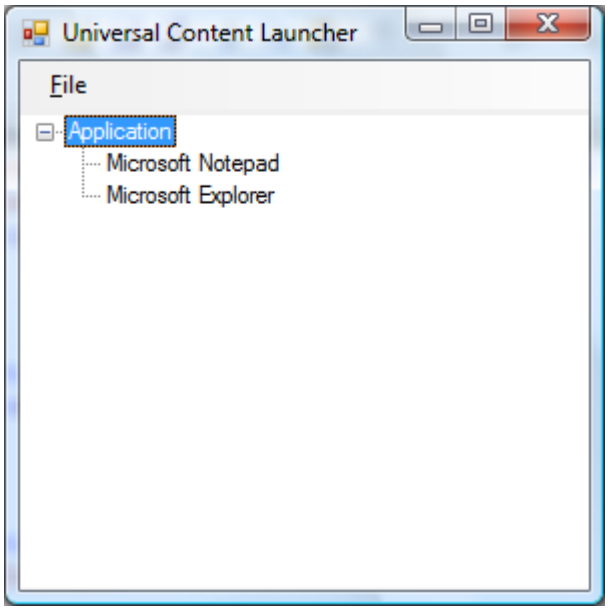
Part One: Expanding the Content Loader

We now have a functioning content launcher that, at this point in time, only launches one application! Let's add another application to our test XML file to check it will display everything correctly when there are multiple selections.

Edit your UCLContent.xml file to add another application to it, however this time, let's enter the path without a trailing backslash, you will see why later! Here's mine, notice the path of `c:\windows` and not `c:\windows\` for the `Microsoft Explorer ExePath` entry:

```
<?xml version="1.0" encoding="utf-8" ?>
<Content>
  <Application>
    <Name>Microsoft Notepad</Name>
    <ExeName>notepad.exe</ExeName>
    <ExePath>c:\windows\</ExePath>
  </Application>
  <Application>
    <Name>Microsoft Explorer</Name>
    <ExeName>explorer.exe</ExeName>
    <ExePath>c:\windows</ExePath>
  </Application>
</Content>
```

Now run it to check that everything is ok:



Wow, we must have done something right! Go ahead and launch your new app, and see what happens, well, unless you have been putting better error trapping into your Content `Launch()` method, then absolutely nothing will happen!

Why is this? Let's debug it a bit:

Set a breakpoint in your `Launch()` method next to the following line:

```
System.IO.Directory.SetCurrentDirectory(this.mExePath);
```

in the UCLContent.cs code and step through the code to see what values in `mExePath` and `mExeName` contain for the following line:

```
System.Diagnostics.Process.Start(this.mExePath + this.mExeName,
this.mArguments);
```

Don't forget that you will need to step through the loop twice, once for each application content, you should have noted the following (using my example applications):

Microsoft Notepad:

```
mExePath = c:\windows\

mExeName = notepad.exe
```

Therefore UCL is looking for: `c:\windows\notepad.exe`

Microsoft Explorer:

```
mExePath = c:\windows

mExeName = explorer.exe
```

Therefore UCL is looking for: `c:\windowsexplorer.exe` which is not valid!

So, as suspected in the last tutorial, we really need to check if the file is valid before trying to launch it so we can pop up a suitable error message. We also need to take into account the fact that the information we have may not be consistent, so people may enter a path with or without a backslash.

The reality of this is that when our project is finished we will be populating the XML content file ourselves from UCL, so we could control the output and make sure that `mExePath` always has a trailing backslash. But this is still open to abuse as the user can still edit the XML file manually if they want to, so we need to deal with this.

Luckily in the .NET framework, there is a really neat way to handle missing backslashes when processing paths and filenames:

```
Path.Combine(first string , second string)
```

What this does is take the strings passed to it, and return a valid path, with all the backslashes in all the correct places, whether they are present in the source data or not. So it no longer matters how the information is entered into our XML file!

Let's cleanup our `Launch()` method by adding two new using statements:

```
using System.IO;
using System.Windows.Forms;
```

To the top of the UCLContent code and then adding in the above fix, plus some file checking. Here's my final Launch() method:

```
        public Boolean Launch()
         {

             try
             {


                 if (File.Exists(Path.Combine(this.mExePath,
this.mExeName)))
                 {

                     Directory.SetCurrentDirectory(this.mExePath);

System.Diagnostics.Process.Start(Path.Combine(this.mExePath ,
this.mExeName), this.mArguments);
                     return true;
                 }
                 else
                 {
                     MessageBox.Show("Cannot locate " + this.mExeName + "!
Has it been removed?");
                     return false;
                 }

             }

             catch (Exception e)
             {
                 Console.WriteLine(e.Message);
                 return false;
             }

         }
```

Notice that our `Launch()` method now traps for a file no longer being present. Therefore we could at a later date, not only prompt the user about the missing file, but also give them the option to remove it from the UCL list at the same time. That would be neat!

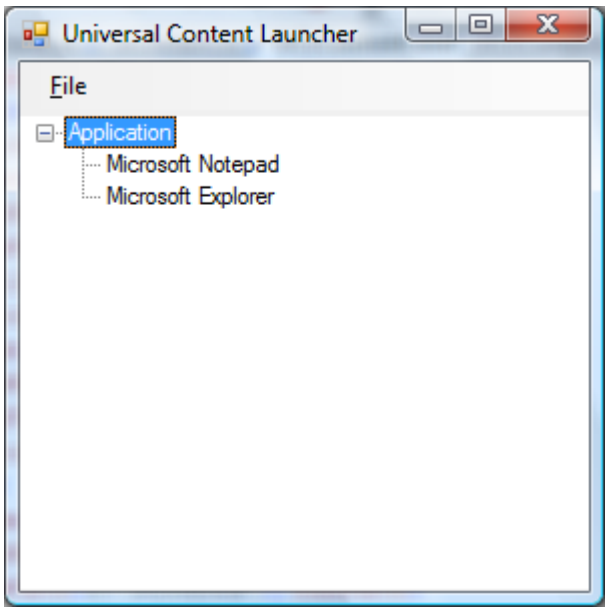
If you run UCL now, the “Windows Explorer” entry launches correctly, despite the fact that we didn’t edit the XML file to show a correct trailing backslash for the path entry.

Well done, now that’s sorted, let’s move on.

Ok, so we’ve added two applications to our content list, and fixed it up to be a bit more robust, now let’s add a game to the XML file, it can actually be anything for the purposes of testing, except that it will be contained in a `<Game>` element rather than an `<Application>` element. I’ve actually used Notepad again with a different heading, as it loads quickly for testing and the altered heading will prove that UCL is looking at the right Notepad entry. Here’s my XML file once more:

```
<?xml version="1.0" encoding="utf-8" ?>
<Content>
  <Application>
    <Name>Microsoft Notepad</Name>
    <ExeName>notepad.exe</ExeName>
    <ExePath>c:\windows\</ExePath>
  </Application>
  <Application>
    <Name>Microsoft Explorer</Name>
    <ExeName>explorer.exe</ExeName>
    <ExePath>c:\windows</ExePath>
  </Application>
  <Game>
    <Name>Microsoft Notepad Games</Name>
    <ExeName>notepad.exe</ExeName>
    <ExePath>c:\windows\</ExePath>
  </Game>
</Content>
```

Ok, let's run it again and see what happens:



Were you expecting that? No "Game" category displayed? You should have been.

Cast your mind back to our `LoadContentList` method. You may recall this line:

```
string tType = "Application";
```

It's pretty obvious what our problem is, we hard coded UCL to look at `"Application"`, with the intention of storing that value in a `string` so that later on in the project, we could change our method to iterate through all the different `UCLType` values and display all of them. Well the time has come to make that change!

So we need another loop to acheive this, and the `foreach` statement fits the bill nicely:

```
                foreach (string tType in
System.Enum.GetNames(typeof(UCLContent.UCLType)))
```

Can you see what we have done here? We have actually included the initialization of `string tType` in our `foreach` loop. So `tType` is defined as a `string` and will contain each of the values present in `UCLType` as the loop progresses by using the `System.Enum.GetNames` method to return their names.

By defining `tType` in this loop, then we don't need to change any of our existing code, we just need to enclose it in braces to make it part of the `foreach` loop, I have coloured these braces red below, to indicate this:

```
        private void LoadContentList()
        {

            if (File.Exists("UCLContent.xml"))
            {
                XPathDocument doc = new XPathDocument("UCLContent.xml");
                XPathNavigator nav = doc.CreateNavigator();

                treeView1.Nodes.Clear();

                TreeNode tNode;

                foreach (string tType in
System.Enum.GetNames(typeof(UCLContent.UCLType)))
                {
                    //string tType = "Application";
                    tNode = treeView1.Nodes.Add(tType);

                    XPathExpression expr;
                    expr = nav.Compile("/Content/" + tType + "/Name");
                    XPathNodeIterator iterator = nav.Select(expr);

                    try
                    {
                        while (iterator.MoveNext())
                        {

                            UCLContent tContent = new UCLContent();

                            XPathNavigator nav2 = iterator.Current.Clone();
                            Type type = tContent.GetType();
                            PropertyInfo tTemp =
type.GetProperty(nav2.Name);
                            tTemp.SetValue(tContent, nav2.Value, null);
                            tNode.Nodes.Add(tContent.Name);

                            tTemp = type.GetProperty("Type");
                            UCLContent.UCLType tUCLType =
(UCLContent.UCLType)Enum.Parse(typeof(UCLContent.UCLType), tType, true);
                            tTemp.SetValue(tContent, tUCLType, null);

                            while (nav2.MoveToNext())
                            {
                                tTemp = type.GetProperty(nav2.Name);
                                tTemp.SetValue(tContent, nav2.Value, null);
                            }

                            UCLContentList.Add(tContent);

                            tContent = null;
```

```
                        }
                    }
                    catch (Exception ex)
                    {
                        Console.WriteLine(ex.Message);
                        MessageBox.Show("Cannot load the UCL content list.
There is a problem processing the UCLContent.xml file!");
                    }
                }
            }
            else
            {
                MessageBox.Show("Cannot load the UCL content list. There is
no UCLContent.xml file present in the UCL folder!");
            }

        }
```

Before we go further, please notice the following line in the code above:

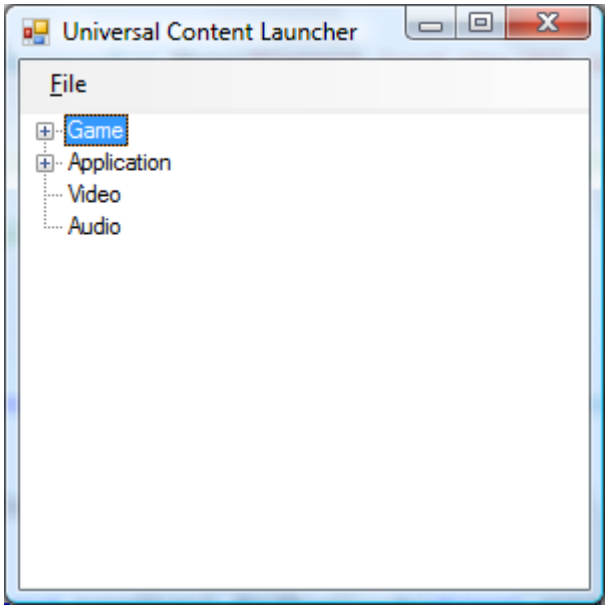
```
                    //string tType = "Application";
```

This is what is known as a comment, you can comment something out by adding `//` to the very beginning of the line. What a comment does, is tell C# to ignore the rest of the line, so in this case we have kept the old piece of code but commented it out so that it is ignored. If our new code goes well, then we can remove this line, if it doesn't, then we still have the old code to refer to, or even roll back to if need be.

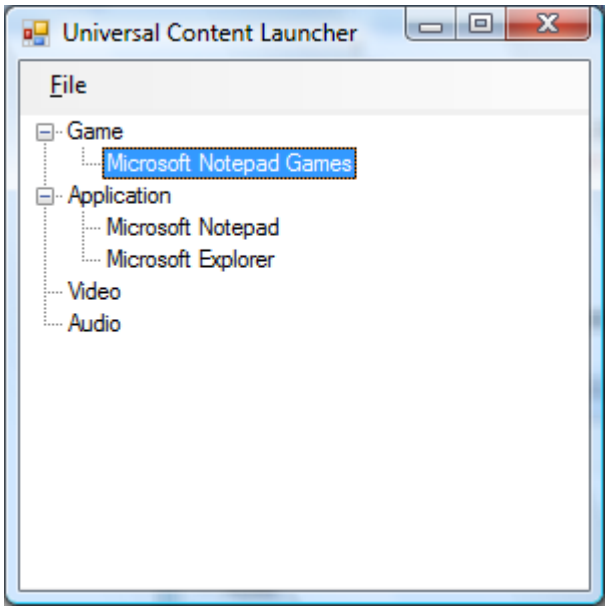
You can also use comments to add text notes describing what your program does, or is doing at a certain point in the code, this will be of help to not only other coders who you may choose to share the source code with, but also yourself, when revisiting the program if you haven't seen it for a while.

You can also comment out multiple lines of text, rather than just a single line, by using `/*` at the beginning of the text and `*/` at the end. I'm traditionally bad a commenting code, so I wont be doing much in these tutorials, but if you work through something and feel you need to add a comment to remind you of what it does, then please do so.

Now if we run UCL, we should see the following:



Wow, we're good, aren't we? Note that we are now showing all the different "Types" regardless of whether or not we have content defined for them in the XML file. Some people like this method, some people would rather hide categories that contain no content. For now, we're going to leave it as it is. Now click on the "+" signs to check the right content is in the right category, and that all of them launch ok.



Yep, it's official, we are good! Now, some people may want to click all those "+" symbols, and some people may want UCL to open in its "expanded" state like above. To open "expanded" we can do this:

```
treeView1.ExpandAll();
```

Put that line right at the end of the method, after the last `MessageBox` but before the final closing brace for the whole method. UCL now opens like the screenshot above, and once

more, we could expand on this at a later date to give the user the choice of opening UCL in its “expanded” view.

That all seemed to happen quite quickly, but we still need to be able to Launch Video and Audio types to complete this stage of the launcher, and as we know from earlier discussions, this will require launching both a media player, and a file to play. Let’s have a look at that next, by putting an `<Audio>` entry into our test XML file, here’s what I put into mine:

```
<Audio>
  <Name>My Vision</Name>
  <ExeName>wmplayer.exe</ExeName>
  <ExePath>c:\program files\windows media player\</ExePath>
  <FileName>My Vision.mp3</FileName>
  <FilePath>f:\mp3\</FilePath>
</Audio>
```

We can predict what will happen if we run UCL now, we will just get the media player opening on its default page, as we haven’t yet written any code to tell UCL how to process `FilePath` and `FileName` when it is present. At the most basic level, which is where we should always begin, we can take `FilePath` and `FileName` and build a string to pass via the “Arguments” property of UCLContent. This will work for most media players, so let’s do that!

Modify your `Launch()` method for UCLContent by adding the code below before the following line:

```
                if (File.Exists(Path.Combine(this.mExePath,
this.mExeName)))
```

Here’s the code:

```
                if (this.mFileName != null)
                {
                    if (File.Exists(Path.Combine(this.mFilePath ,
this.mFileName)))
                    {
                        this.mArguments = "\"" +
Path.Combine(this.mFilePath, this.mFileName) + "\"";
                    }
                    else
                    {
                        MessageBox.Show("Cannot locate the file " +
this.mFileName + "! Has it been removed?");
                        return false;
                    }
                }
```

This should all make sense to you, but look at the lines above, remember how I told you way back in tutorial one that we need to enter a `"\"` in a string as `"\\"` for C# to recognise it. Well here we see `"\""` being entered.

The reason why this is, is that the first `"\"` tells C# to process the following character as a literal character. So whatever follows `"\"` will be inserted into the string correctly when our program is run. This is necessary as some characters we enter in a string will actually cause a compiler error before we run the program as they actually change the way the C# compiler reads your program.

Try to enter the above line like this `"""` or this `""""`, and you will see how confused C# behaves until you enter this `"\""`. If you are having compiler errors entering a string, and have some "odd" characters in there, then try inserting a `"\"` before it, it may just work then!

Run UCL, and amazingly your MP3 will now play correctly. The way we entered the code also ensures that video content will work as well. How good are we?

Of course, this solution may have problems with different media players, but sometimes you go with what you know and address issues like that if they arise – no point wasting effort on something that potentially may never be used!

Ok, just to clarify where we are now, here's my final code for the complete `Launch()` method:

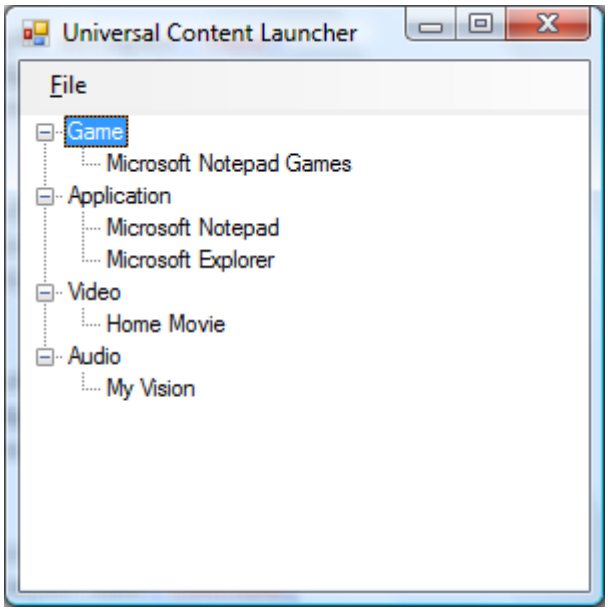
```
        public Boolean Launch()
        {

            try
            {

                if (this.mFileName != null)
                {
                    if (File.Exists(Path.Combine(this.mFilePath ,
this.mFileName)))
                    {
                        this.mArguments = "\"" +
Path.Combine(this.mFilePath, this.mFileName) + "\"";
                    }
                    else
                    {
                        MessageBox.Show("Cannot locate the file " +
this.mFileName + "! Has it been removed?");
                        return false;
                    }
                }


                if (File.Exists(Path.Combine(this.mExePath,
this.mExeName)))
                {

                    Directory.SetCurrentDirectory(this.mExePath);

System.Diagnostics.Process.Start(Path.Combine(this.mExePath ,
this.mExeName), this.mArguments);
                    return true;
                }
```

```
                else
                {
                    MessageBox.Show("Cannot locate " + this.mExeName + "!
Has it been removed?");
                    return false;
                }

            }

            catch (Exception e)
            {
                Console.WriteLine(e.Message);
                return false;
            }

        }
```

Hopefully that should make complete sense to you. To check that our code does indeed run a video file, I have added the following lines to my test XML file:

```
  <Video>
    <Name>Home Movie</Name>
    <ExeName>wmplayer.exe</ExeName>
    <ExePath>c:\program files\windows media player\</ExePath>
    <FileName>Home Movie.avi</FileName>
    <FilePath>e:\camera\</FilePath>
  </Video>
```

Which gives me the following content list in UCL, all of which now launch perfectly.



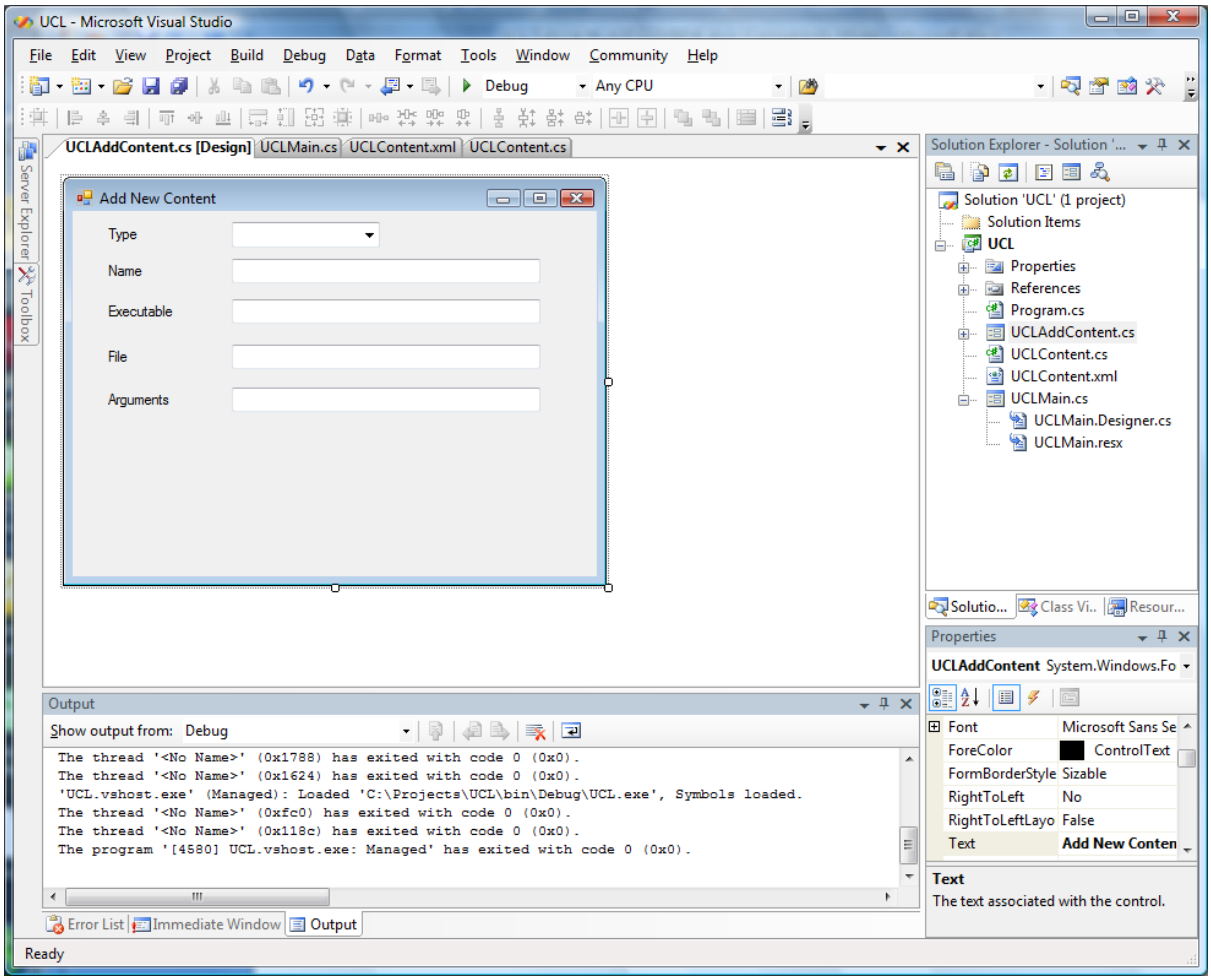
It's interesting to see that we seemed to do a lot of groundwork for this project, typing in a lot of methods and classes that didn't seem to yield any positive results, yet here we are in tutorial four with all of it quickly falling into place, resulting in a feeling that we are almost there!

But there's more to be done. I mean, sure we can display content that we manually edit into our XML file, but we need to give the user an easy way to manage their content list.

Part Two: Creating the Add New Content form

We need a new form, that can allow for all of the available values we can have, to be set from within UCL. So right click on “UCL” in the solution explorer and select Add -> Windows Form...

Call the form UCLAddContent and create a similar form to mine by dragging the appropriate controls onto the form from the toolbox:



Here’s what I did, step by step:

- Created the form
- Called the form “Add New Content” by changing its Text property
- Dragged on one label and then clicked on it to select it, pressed ctrl+c to copy it and then pressed ctrl+v repeatedly to paste it four times, before positioning them correctly. Note that when you position things on forms, you get different coloured “alignment” lines to help you align everything correctly.
- Named the labels Type, Name, Executable, File and Arguments by changing the Text property for each label.
- Dragged on a combobox and called it comboType by changing its Name property.

- Dragged on one text box and set the Anchor property to Left,Top,Right, so it will grow if the form is resized, and then copy and pasted to create four text boxes in total. I then positioned them correctly and named them textName, textExecutable, textFile and textArguments by changing the Name property.

Here's a tip for you: If you create a lot of controls on a form and want to change all of their properties to the same time; so say we created the four text boxes above, and then wanted to change the Anchor property for all four then you can click on the first control, and then shift and click on other controls to group them together. If you then change any of the properties after doing this, then the change will be applied to all of the grouped controls.

Ok, when you have got to the same point, we need to build in a way of displaying the form when UCL is running. As we already have a menu, lets use that!

- Go to your UCLMain form design
- Click on the menu to enter the menus "edit" mode
- To the right of "File" type in "&Edit"
- Below "&Edit" type in "&Add New Content"

Great, now we have a menu option to display the form, let's add in the code to do that, so double click on "&Add New Content" in the menu and you will see this new method:

```
        private void addNewContentToolStripMenuItem_Click(object sender,
EventArgs e)
        {

        }
```

We already know from past experience that this is the correct method we need to use. Now add the following code to the method:

```
            UCLAddContent frmUCLAddContent = new UCLAddContent();
            frmUCLAddContent.Show();
```

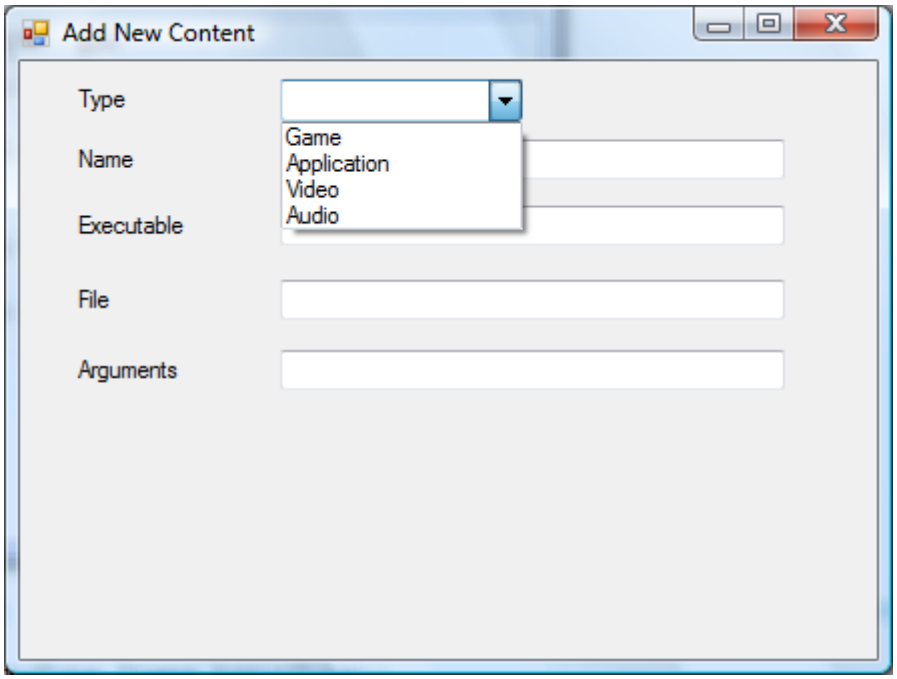
You should be getting familiar with things like this by now. We are declaring a new object called `frmUCLAddContent` that is of type `UCLAddContent` and then setting it to a new instance of `UCLAddContent` (our new form that we just created)

We are then simply showing it, which will display it on the screen. Try it by running UCL and selecting Edit -> Add New Content from the menu. It should have displayed ok, and you will notice that it isn't really doing much at the moment!

The obvious thing that is missing is a lack of a Type wntry in the dropdown to select from, so let's make sure we always populate the Type combobox whenever our form loads by adding the following code to the forms Load() method (can you remember how to add a new form event – clue: has lightning struck twice?):

```
            foreach (string tType in
System.Enum.GetNames(typeof(UCLContent.UCLType)))
            {
                comboType.Items.Add(tType);
            }
```

We've done that before, in the `LoadContentList()` method of the UCLMain form, and note that we add items to a combobox in a similar way that we add nodes to a treeview control, albeit without the parent and child complexities. Is this getting easier or what? Now you should see this when you run UCL and Add New Content:

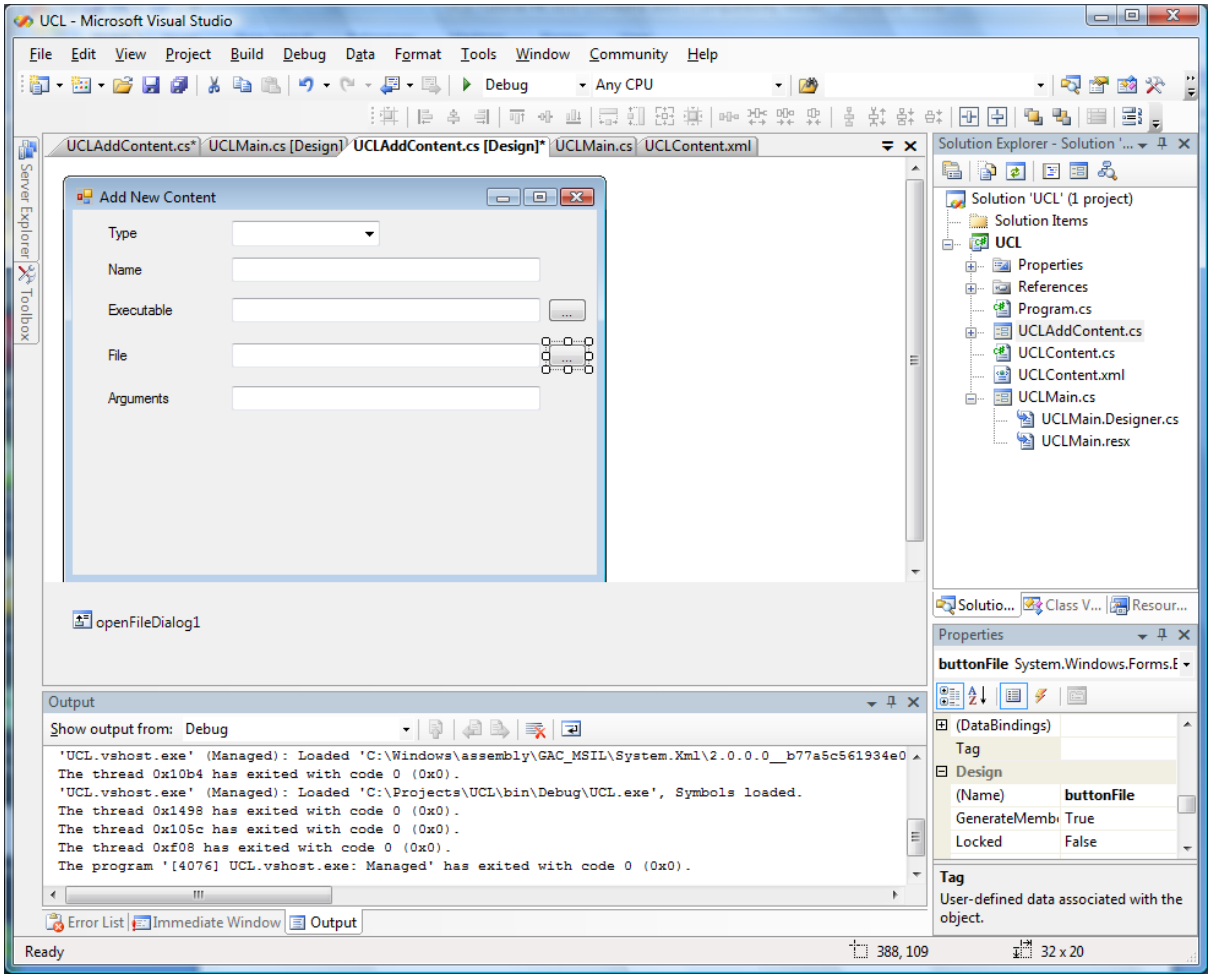


But hang on a minute – are we really going to expect the user to type in a pathname or executable name for the content? And why do we only have one textbox for executable, instead of one for executable path and one for executable name? And the same goes for the file textbox!

All right, calm down! It's ok, I do sometimes know what i'm doing!

We're going to make these selections ultra user friendly by letting the user browse to the file they want. Our textbox is only there for the purpose of displaying the file they browsed to, so they don't forget they have already done that.

Well, if you're worried this is going to be tonnes of work in C#, then you're in luck! It's very easy! But first we need to add a button to allow the user to select a file, so do something similar to the following screenshot, and call them buttonExecutable and buttonFile:



Note:

- The "..." on the buttons is achieved by simply setting the Text property to "..."
- The button Anchor should be set to Top,Right or else things will get messy on form resizing – check out different Anchor settings to see what i mean!

Now doubleclick on the button and add in the code we need:

```
        private void buttonExecutable_Click(object sender, EventArgs e)
        {

            OpenFileDialog newFile = new OpenFileDialog();

            newFile.Title = "Please select the content to launch";
            newFile.Filter = "Windows Executable
(Exe,Com,Bat)|*.exe;*.bat;*.com";
            if (newFile.ShowDialog() == DialogResult.OK)
            {
                textExecutable.Text = newFile.FileName;
            }

        }
```

Hopefully you can start to see what the code I just pasted in does, before I tell you?

```
OpenFileDialog newFile = new OpenFileDialog();
```

This line creates a new `OpenFileDialog` of the name `newFile`.

```
newFile.Title = "Please select the content to launch";
```

This line defines what the title of our `OpenFileDialog` will be when we display it.

```
newFile.Filter = "Windows Executable
(Exe,Com,Bat)|*.exe;*.bat;*.com";
```

This line can be tricky. It shows the filter we are going to apply to the files that our `OpenFileDialog` displays. To the left of the `|` is the text that will be displayed in the `OpenFileDialog` file type selector, to the right are the actual filters the `OpenFileDialog` will apply when showing files.

So in this example only files of type exe, com and bat can be displayed and selected with this `OpenFileDialog`. We are therefore limiting what the user can select and by doing that, minimising the risk of an invalid entry.

```
if (newFile.ShowDialog() == DialogResult.OK)
```

This line actually does 2 things in one `if` statement! Firstly it shows our dialogue, and secondly it checks to see if the response back from it is ok (i.e. the user selected a file). This means our program will wait until the dialogue has completed before continuing.

```
textExecutable.Text = newFile.FileName;
```

And this line actually records the result in our `textExecutable` text box.

Try it for yourself.

Can you see any problems here? We get a returned file path and executable in a single string, but we made the decision to store them as seperate entities in our XML file. We need a `Path.UnCombine` function!

Well, there's no such thing – but if I told you there is `Path.GetDirectoryName()` and `Path.GetFileName()` then that should put your mind to rest.

Now we need to do the same for the File text box, however, there are a number of issues with this box:

- We only really need to display it for Video and Audio Content
- The File filters we use will differ according to the content we have selected.

Oh dear – we need to change something. I mean, we could hard code all this for now, but in the future we may need to expand on it dynamically. For example imagine if I wanted to launch Microsoft Notepad and pass it a text file to open.

But it gets worse – what if I defined an Audio file as type MP3\WMA\WAV, and someone comes along who wants to play OGG – It's just not good enough to second guess all these possibilities, we need a way to store this information and let the user change it.

Well, we've thought about it, but for now it's out of scope. When we design the user settings functionality, then we can address it properley. If we try and second guess what we are going to do later, then we will get ourselves tied up in knots, and probably only end up changing it again anyway. It's sometimes better to make sure you finish something basic to build on, rather then get half way through an all singing, all dancing masterpiece and lose interest!

So for now we are indeed going to hard code this, and change it later, and this will give us a good insight into how to make design changes at a later date. If we didn't do this, then this tutorial would be very, very long!

The good thing about hard coding this, is that it introduces us to another C# statement, and also, don't forget that most of this code will be re-useable anyway. Here's the Method for the File button:

```
        private void buttonFile_Click(object sender, EventArgs e)
        {

            OpenFileDialog newFile = new OpenFileDialog();

            newFile.Title = "Please select the file to play";

            switch(comboType.Text){
                case "Video":
                    newFile.Filter = "Video File
(MPG,WMV,AVI)|*.mpg;*.wmv;*.avi";
                    break;
                case "Audio":
                    newFile.Filter = "Audio File
(MP3,WMA,WAV)|*.mp3;*.wma;*.wav";
                    break;
                default:
                    MessageBox.Show("You can only select a file to play for
Audio and Video content!");
                    return;
            }

            if (newFile.ShowDialog() == DialogResult.OK)
            {
                textFile.Text = newFile.FileName;
            }

        }
```

As you can no doubt see, this is pretty much the same as the Executable File method, but with one exception:

```
            switch(comboType.Text){
```

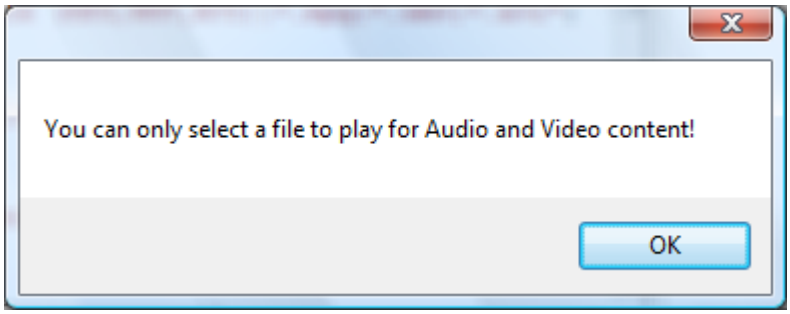
What the `switch` statement does, is allow your code to branch depending on a choice. It is similar to an `if` statement, but is a lot easier to read. In the switch statement you define what it is that you are going to compare, so in this case we are looking at the Type name that is selected in the Type combo box on our form.

After the switch you enter a number of `case` statements and the code to execute if that `case` statement is true. So you can see from the above code that we specifically identify `"Audio"`

and `"Video"` and set the `OpenFileDialog` filter accordingly.

Note that each `case` must be terminated by a `break` statement, or C# will error. The `break` statement simply tells C# that the end of the `case` code has been reached, and to skip over the other `case` selections.

Note also that at the end of the `case` statement we include a `default` statement, this is the code that will run if none of the `case` statements are matched. In the above example, if the Type combo box on our form does not contain `"Audio"` or `"Video"` than a messagebox will be displayed:



And the method terminated.

Yes there are a lot of potential future problems with this code, there always will be when you hard code something, as you are creating something that is only gauranteed to work at that point in time, but for now, let's go with it!

We now have a basic form for entering the required information into our XML file, so how do we now get it in there after we have populated the form?

Part Three: Adding records to the XML Content file

So, how do we add a new record to our XML file?

Well, put a new button on your form, give it the Name “buttonAddContent” and set its Text property to “Add”. Put this at the bottom right of the form, and therefore set its Anchor to Bottom,Right.

Now double click on it to enter the code. Before we begin, add the following using statements to the top of your code:

```
using System.Xml;
using System.IO;
```

Here is the code we will be inserting, see if you understand some of it before we go through it all:

```
        private void buttonAddContent_Click(object sender, EventArgs e)
        {
            if (comboType.Text == "")
            {
                MessageBox.Show("Please select a Type for this Content!");
                return;
            }

            if (textName.Text == "")
            {
                MessageBox.Show("Please select a Name for this Content!");
                return;
            }

            if (textExecutable.Text == "")
            {
                MessageBox.Show("Please select the Executable File for this
Content!");
                return;
            }

            XmlDocument doc = new XmlDocument();
            doc.Load("UCLContent.xml");

            XmlElement elementType = doc.CreateElement(comboType.Text);

            XmlElement elementNode = doc.CreateElement("Name");
            elementNode.InnerText = textName.Text;
            elementType.AppendChild(elementNode);

            elementNode = doc.CreateElement("ExeName");
            elementNode.InnerText = Path.GetFileName(textExecutable.Text);
            elementType.AppendChild(elementNode);

            elementNode = doc.CreateElement("ExePath");
            elementNode.InnerText =
Path.GetDirectoryName(textExecutable.Text);
            elementType.AppendChild(elementNode);

            if (textFile.Text != "")
            {
```

```
                elementNode = doc.CreateElement("FileName");
                elementNode.InnerText = Path.GetFileName(textFile.Text);
                elementType.AppendChild(elementNode);

                elementNode = doc.CreateElement("FilePath");
                elementNode.InnerText =
Path.GetDirectoryName(textFile.Text);
                elementType.AppendChild(elementNode);

            }

            if (textArguments.Text != "")
            {
                elementNode = doc.CreateElement("Arguments");
                elementNode.InnerText = textArguments.Text;
                elementType.AppendChild(elementNode);

            }

            doc.DocumentElement.AppendChild(elementType);
            doc.Save("UCLContent.xml");

        }
```

It looks quite long, but is actually pretty simple, for a change!

First we need to make sure the minimum required fields are populated before we attempt to write our record to the XML file, otherwise we are simply asking for trouble. We do this in these lines of code:

```
            if (comboType.Text == "")
            {
                MessageBox.Show("Please select a Type for this Content!");
                return;
            }

            if (textName.Text == "")
            {
                MessageBox.Show("Please select a Name for this Content!");
                return;
            }

            if (textExecutable.Text == "")
            {
                MessageBox.Show("Please select the Executable File for this
Content!");
                return;
            }
```

That should make perfect sense to you now!

Next we open the XML file:

```
            XmlDocument doc = new XmlDocument();
            doc.Load("UCLContent.xml");
```

```
            XmlElement elementType = doc.CreateElement(comboType.Text);
```

Here we are creating a new XML element called `elementType` for the `doc` object (our XML file), this element will have the same name as whatever we selected in the Type combo box, so at the moment it could be either “Game”, “Application”, “Audio” or “Video”

```
            XmlElement elementNode = doc.CreateElement("Name");
```

Now, we have created another element called `"Name"`.

```
            elementNode.InnerText = textName.Text;
            elementType.AppendChild(elementNode);
```

We then assign the new element the value that is in our textName text box, and then append this element to our original `elementType` object. We are essentially creating the parent child structure again, but this time for the XML file rather than for our treeview control.

```
            elementNode = doc.CreateElement("ExeName");
            elementNode.InnerText = Path.GetFileName(textExecutable.Text);
            elementType.AppendChild(elementNode);

            elementNode = doc.CreateElement("ExePath");
            elementNode.InnerText =
Path.GetDirectoryName(textExecutable.Text);
            elementType.AppendChild(elementNode);
```

We then do exactly the same for the elements `"ExeName"` and `"ExePath"`, the two things to note here are that firstly we are continually re-using the `elementNode` object, as once we have appended it to the `CreateElement` object, then it is disposable. Secondly we are using those two `Path` functions we spoke of earlier to split our Executable string into `"ExeName"` and `"ExePath"`.

```
            if (textFile.Text != "")
            {
                elementNode = doc.CreateElement("FileName");
                elementNode.InnerText = Path.GetFileName(textFile.Text);
                elementType.AppendChild(elementNode);

                elementNode = doc.CreateElement("FilePath");
                elementNode.InnerText =
Path.GetDirectoryName(textFile.Text);
                elementType.AppendChild(elementNode);

            }
```

Again, this is exactly the same thing for the File string, the only difference is that because we consider this an optional element, then we check for a valid entry first, and simply don’t create the elements for the XML file if there isn’t one.

```
            if (textArguments.Text != "")
            {
                elementNode = doc.CreateElement("Arguments");
                elementNode.InnerText = textArguments.Text;
                elementType.AppendChild(elementNode);

            }
```

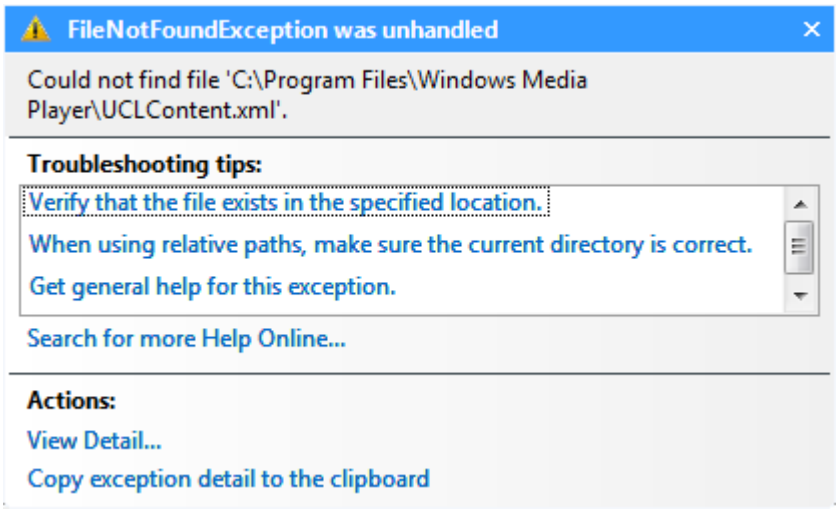
And again, for the optional `"Arguments"` element.

```
            doc.DocumentElement.AppendChild(elementType);
            doc.Save("UCLContent.xml");
```

And finally we append our new element `elementType` and all it's child elements into our XML file, before saving it.

Simple heh, type it in, run UCL and be prepared for a dissapointment!

Enter something valid (select at least a Type, Name and Valid Executable) in our Add New Content form before trying to save it with the "Add" button:



Not fair!

In the example above I was simply trying to select the windows media player and enter it as a stand alone application in my XML file. Look at the error above, for some reason the path that I navigated to is being used by UCL as the location where it thinks our XML file now resides?

This is because of this again:

```
            doc.Load("UCLContent.xml");
```

There is no path defined for the XML file, but we sorted that didn't we? Well, we sorted it for the last problem, but this one is different as the `OpenFileDialog` appears to have changed the computers current directory to the last one we visited. So when we try to open the XML file, UCL is looking in the current directory, which has changed now and is no longer the UCL directory!

We need to make this foolproof, and luckily we can by changing this line:

```
            doc.Load("UCLContent.xml");
```

To this:

```
            doc.Load(Path.Combine(Application.StartupPath,
"UCLContent.xml"));
```

Note how we are using `Path.Combine` again, and the `Application.StartupPath` object returns the path that our application initially ran from.

We also need to alter the code where we save the XML file to this:

```
            doc.Save(Path.Combine(Application.StartupPath,
"UCLContent.xml"));
```

And while your at it, maybe you should change the original code in the `LoadContentList()` method in UCLMain.cs to do the same thing:

```
            if (File.Exists(Path.Combine(Application.StartupPath,
"UCLContent.xml")))
            {
                XPathDocument doc = new
XPathDocument(Path.Combine(Application.StartupPath, "UCLContent.xml"));
```

Good, that feels safer!

Now run UCL and you will be able to add new elements to the XML file perfectly, but there are still some problems:

- We should tell the user the operation was succesfull
- We should exit the form after a succesfull save, as its purpose is fulfilled.
- We need to refresh the Content treeview list, so our new content is visible!

The first two problems are easily solved by adding these two lines to the bottom of the `buttonAddContent_Click` method. So after this line:

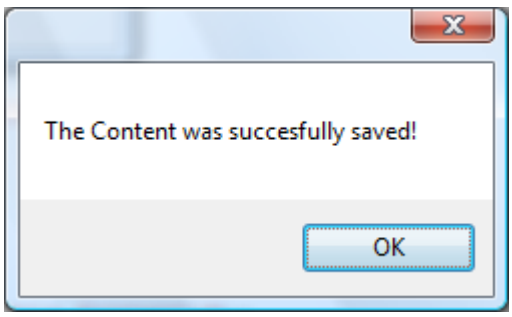
```
            doc.Save(Path.Combine(Application.StartupPath,
"UCLContent.xml"));
```

Add:

```
            this.Dispose();
            MessageBox.Show("The Content was succesfully saved!");
```

We know about `MessageBox`, and the method `this.Dispose()` simply tells C# to close or dispose of the object the code sits within (`this`).

Adding new content now will result in the form closing and the following message box being displayed:



Lastly to refresh our Content List, it would be nice if we can do this:

```
UCLMain.LoadContentList();
```

However, if you try this, it won't work!

The reason it wont work is that the code above is referencing the object `UCLMain`, this is not the same as our form UCLMain. The problem is that the form UCLMain is an instance of the object `UCLMain`, so `UCLMain` is like the template that created the form.

This means that if you called a method or accessed a property of `UCLMain` then you would not be looking at the values held in the UCLMain form! We need to somehow tell our "Add New Content" form exactly what object it was opened from, so that we can communicate back to it.

If you know a little about programming classes, then you may be wondering why we can't just use the `static` keyword for the method, as this would usually enable the method to be called in this way. However, this is not applicable here, because a `static` method is a method that can be called correctly from the `UCLMain` object, whereas the `LoadContentList()` method is relevant to the instance of the `UCLMain` object and not the `UCLMain` object itself. (i.e. our form)

Well, if you followed all that, then well done! The way we need to do this is to modify the way we initially opened the "Add New Content" form. We need to change the following line of code in our `addNewContentToolStripMenuItem_Click` method in UCLMain, from this:

```
UCLAddContent frmUCLAddContent = new UCLAddContent();
```

To this:

```
UCLAddContent frmUCLAddContent = new UCLAddContent(this);
```

This will then pass the UCLMain form object to our Add New Content form as it is loaded. In order for this to work we need to also declare a new variable in the "Add New Content" form that can be used in all of its own methods. Can you remember how to do this?

If you can't you need to add this line after the class line for the form, as shown below:

```
public partial class UCLAddContent : Form
{
    private UCLMain frmUCLMain;
```

Then we need to store the passed form object in our `UCLMainForm` variable, by altering the `UCLAddContent` method that initialises our form, like so:

```
public UCLAddContent(UCLMain OpenedBy)
{
    InitializeComponent();
    frmUCLMain = OpenedBy;
}
```

We now hold a local reference to the form that opened our "Add New Content" Form. As this reference was declared outside of any method, then we can also use it in our save button

method, by adding the following line, after the `MessageBox.Show("The Content was succesfully saved!");` line of code:

```
frmUCLMain.LoadContentList();
```

Go on – try it – it works!

Wow, that was quite a ride, but we've now got a more complete Launcher, and can even Add new content to our XML file!

In the next tutorial we will be addressing the issue of both modifying existing content, and removing unwanted content. We will also tidy up some other bits and bobs to leave our Launcher in a completed state.

We can then move on to more advanced topics, and start to add a bit of polish to our launcher, as well as expand its capabilities and address some of the issues we have "coded around" during this tutorial.